Представьте, что имеется следующая программа на языке C++:

```cpp
int main()
{
    int b {3};
    int a = 7 - b;

    return 0;
}
```

Здесь a, b – обычные целочисленные переменные, расположенные где-то в памяти устройства. Про такие переменные можно сказать, что они являются леводопустимыми (lvalue) выражениями, то есть, выражениями, которым можно присваивать значения определенного типа.

Конечно, простые переменные – это частный случай леводопустимых выражений. Другой пример – это указатель на выделенную область памяти:

```cpp
    double* ptr_b = new double;
    *ptr_b = 6.43;
    delete ptr_b;
```

Или доступ к отдельным элементам массива:

```cpp
    char str[] = "Hello";
    str[0] = 'D';
```

Или ссылка на тот или иной тип данных:

```cpp
    int& count = b;
```

Или объект класса (структуры):

```cpp
class Point {
public:
    int x{0}, y{0};
};
…
Point pt = {1, 2};
```

Или даже метод (функция), который возвращает конструкцию, связанную с областью памяти:

```language-cpp
class Point {
public:
    int x{0}, y{0};
public:
    int& get_x() { return x; }
};
…
    Point pt = {1, 2};
    pt.get_x() = 10;
```

также является леводопустимым.

Характерной особенностью всех lvalue выражений является то, что они связаны прямо или косвенно с областью памяти, в которой хранятся данные определенного типа. И часто эти данные можно изменить (присвоить им другие значения), если конечно, нет ограничений модификатора const.

Выражения rvalue

Однако не все выражения в программе можно воспринимать, как lvalue. Например, нельзя целочисленной константе присвоить какое-либо другое значение:

```language-cpp
int main()
{
    int b {3};
    int a = 7 - b;

    5 = a; // ошибка

    return 0;
}
```

Такие величины относятся к rvalue выражениям. Или, если взять две переменные и попытаться что-либо присвоить их сумме:

```language-cpp
a + b = 10;
```

то будет ошибка, так как каждая из переменных – это lvalue выражение, но их сумма уже не связана с какой-либо постоянной областью памяти, в которую можно занести новое значение, а потому относится к rvalue выражению. Даже обычный унарный плюс перед переменной:

```language-cpp
+b = 10;
```

превращает ее в rvalue выражение.

То же самое будет наблюдаться и с любыми другими lvalue выражениями:

```language-cpp
int main()
{
    int b {3};
    int a = 7 - b;

    short ar[5] {0};
    ar[0] + b = 5;      // ошибка

    int* ptr_a = &a;
    int& lnk_b = b;

    (lnk_b * 5) = 10;       // ошибка
    (*ptr_a + b) = 10;      // ошибка

    return 0;
}
```

Одно из немногих исключений – это операция разыменывания адреса указателя:

```language-cpp
*(ptr_a + b) = 10;      // ok
```

Но это уже относится к адресной арифметике, и ничего необычного в этом нет.

Ссылки на lvalue и rvalue выражения

Почему в языке C++ так важно различать lvalue и rvalue выражения? Одна из причин, чтобы мы корректно в программах формировали конструкции для присвоения тех или иных значений в допустимые области памяти. Другая причина состоит в том, что только на lvalue выражения можно формировать знакомые нам ссылки. Например:

```language-cpp
int main()
{
    int b {3};
    int a = 7 - b;
```

```
    Point pt;

    int& lnk_a = a;
    Point& lnk_pt = pt;

    return 0;
}
```

Соответственно, на rvalue выражения ссылки вести не могут, кроме константных:

```language-cpp
int main()
{
    int b {3};
    int a = 7 - b;

    Point pt;

    int& lnk_a = 7 - b;         // rvalue
    Point& lnk_pt = Point();    // rvalue

    const int& lnk_a_cnst = 7 - b;         // ok
    const Point& lnk_pt_cnst = Point();    // ok

    return 0;
}
```

Однако, начиная со стандарта C++11, появился новый тип ссылок для rvalue выражений, который записывается с двумя амперсандами следующим образом:

```language-cpp
int main()
{
...
    int&& lnk_a = 7 - b;         // rvalue
    Point&& lnk_pt = Point();    // rvalue
...
    return 0;
}
```

При этом временные объекты, на которые они ссылаются, продолжают существовать, пока существуют на них эти ссылки. Соответственно, через такие ссылки мы совершенно спокойно можем менять состояние объектов или выражений:

```cpp
  std::cout << lnk_a << " " << lnk_pt.x << std::endl;
    lnk_a = 10;
    lnk_pt.x = 5;
    std::cout << lnk_a << " " << lnk_pt.x << std::endl;
```

Конечно, то же самое мы могли бы сделать и через константные ссылки, но, во-первых, это было бы неправильно, т.к. убирать модификатор const – часто порочная практика, а, во-вторых, **в C++ имеются специальные конструкции, которые работают исключительно со ссылками на rvalue выражения**, о которых речь пойдет на следующем занятии.

Функция std::move

Обратите внимание, что rvalue-ссылкам нельзя напрямую присваивать lvalue-ссылки и вообще любые lvalue выражения:

```cpp
int main()
{
    int b {3};
    int a = 7 - b;
...
    int &lnk_la = a;
    int* ptr_a = &a;

    int&& lnk_ra_1 = a;          // ошибка
    int&& lnk_ra_2 = *ptr_a;  // ошибка
    int&& lnk_ra_3 = lnk_a;  // ошибка

    return 0;
}
```

Однако мы можем обойти это ограничение, если воспользоваться операцией преобразования типов, либо специальной функцией std::move следующим образом:

```cpp
    int&& lnk_ra_1 = static_cast<int&&>(a);
    int&& lnk_ra_2 = std::move(*ptr_a);
    int&& lnk_ra_3 = std::move(lnk_a);
```

На практике предпочтение следует отдавать функции std::move, т.к. она, по сути, является надстройкой над оператором static_cast<T&&> и обеспечивает более безопасное преобразование типов ссылок.

А вот обратное присвоение lvalue ссылкам rvalue ссылок можно делать без каких-либо операций преобразования типов:

```language-cpp
    lnk_la = lnk_ra_1;
    *ptr_a = lnk_ra_1;
```

Итак, из этого занятия вы должны четко различать lvalue и rvalue выражения, и знать, как формируются ссылки на rvalue выражения.